cin.ufpe.br

Centro de Informática

U · F · P · E

Universidade Federal de Pernambuco

Projetando Sistemas Digitais com SystemVerilog

Edna Barros

Grupo de Engenharia da Computação

Centro de Informática -UFPE

Agenda

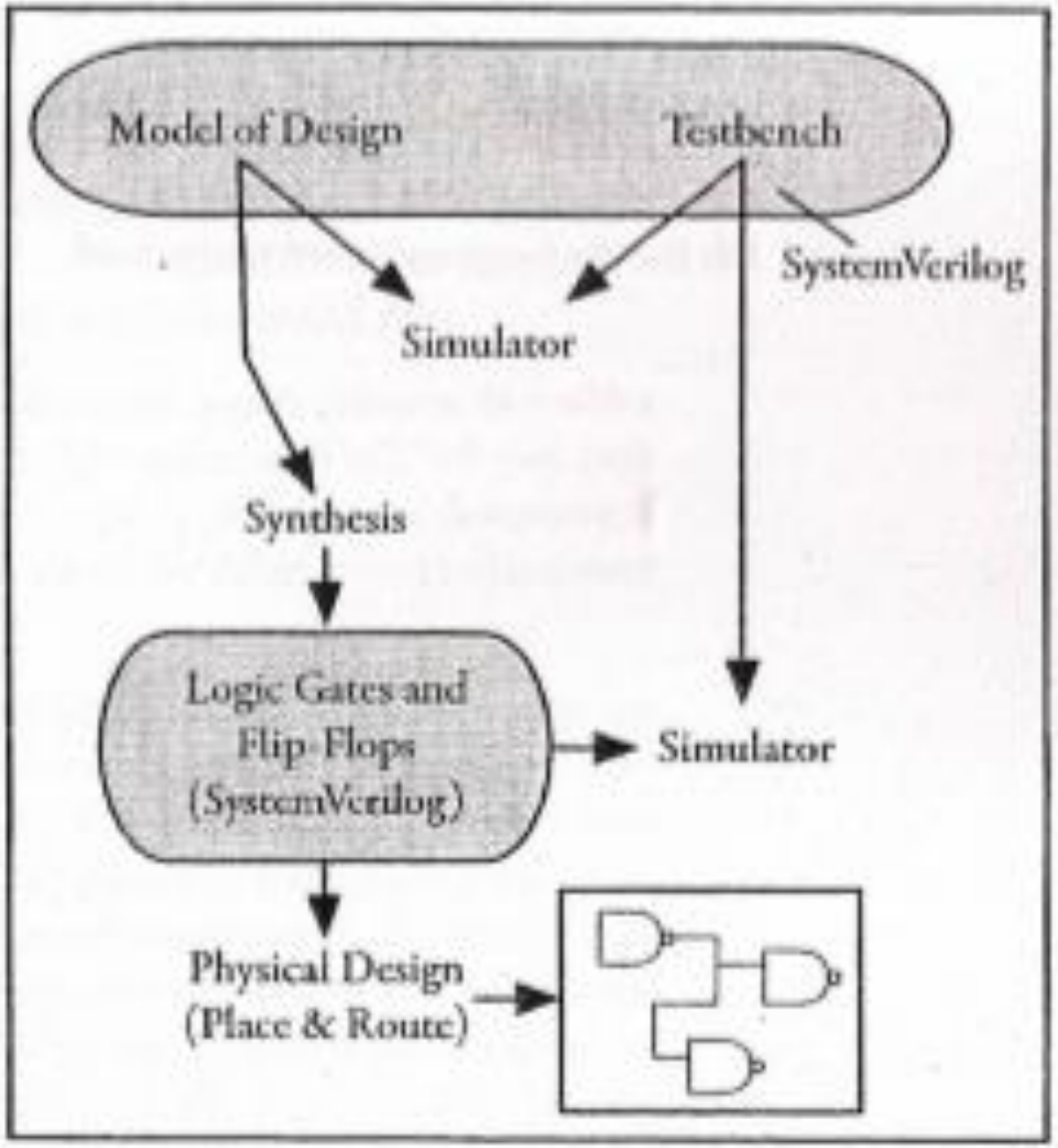
- RTL Design
- Simulando circuitos digitais
- Descrevendo Testbenches
- Módulos parametrizados
- Especificando Portas
- Modelando Circuitos Combinacionais
- Circuitos sequenciais
- Máquinas de estado
  - Modelos esquemático e RTL

RTL Design

- Sistemas digitais possuem milhões de portas lógicas e transistores
  - Necessidade de ferramentas EDA (Electronic Design Automation)
  - Especificação em linguagem específica (HDL – Hardware Description Language)
  - Nível de abstração – RTL (Register Transfer Level)
    - Circuitos combinacionais
    - Registradores
    - Máquinas de estado (controle)

Fluxo de Projeto

- Nível de abstração – RTL (Register Transfer Level)
  - Circuitos combinacionais
  - Registradores
  - Máquinas de estado (controle)



Simulador

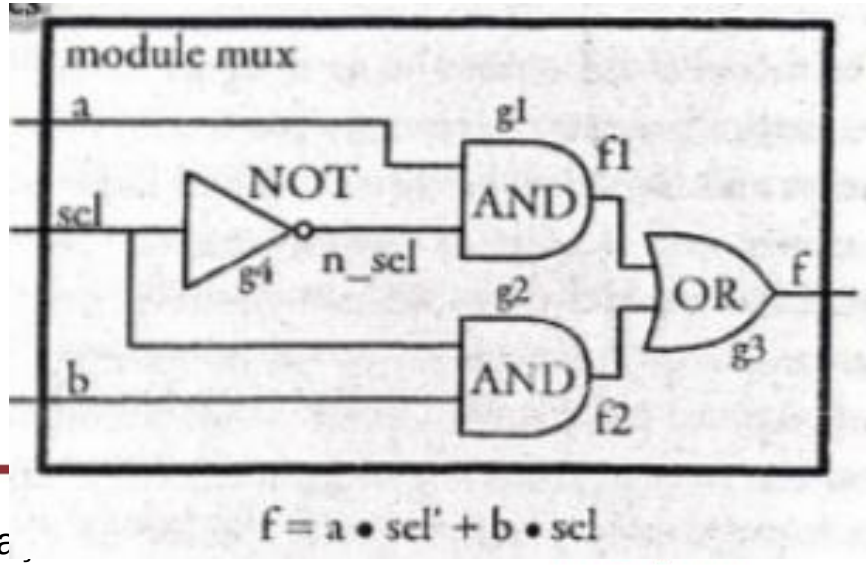
- Programa que prediz o comportamento de um sistema físico em função do tempo
  - Previsão climática: simulador do comportamento do clima
  - Modelagem em função do tempo de um sistema digital

Simulador

- SystemVerilog
  - Linguagem que modela um sistema digital
    - Por exemplo: portas lógicas conectadas
- Simulador que executa o comportamento do sistema modelado em SystemVerilog
  - Como os valores das saídas variam em função dos valores das entradas no tempo
  - Permite verificar se o modelo do sistema digital implementa a função desejada

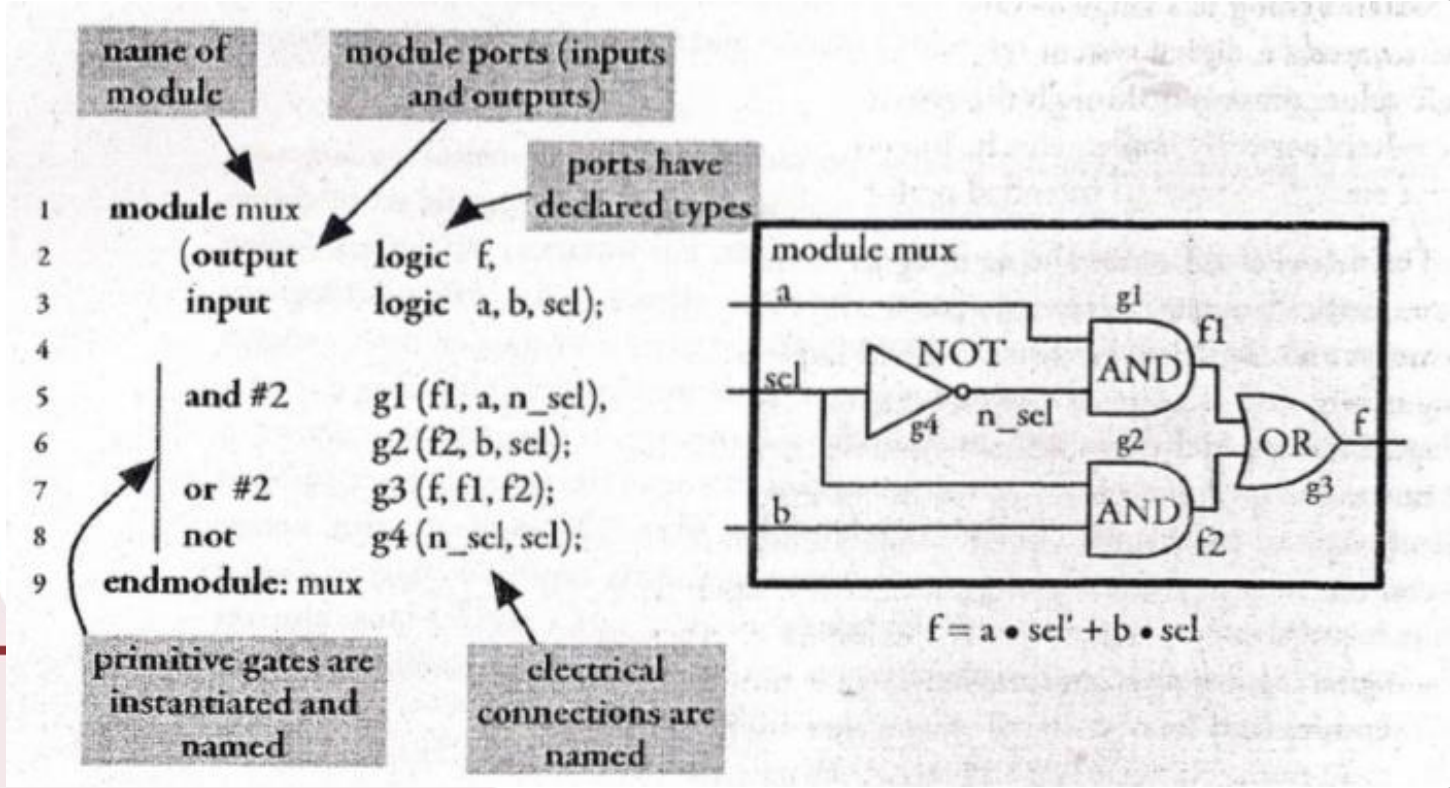
SystemVerilog

- Permite modelar um sistema digital considerando:
  - Modelo da interconexão
  - Modelo de tempo
  - Modelo da funcionalidade
- Modelagem em vários níveis de abstração



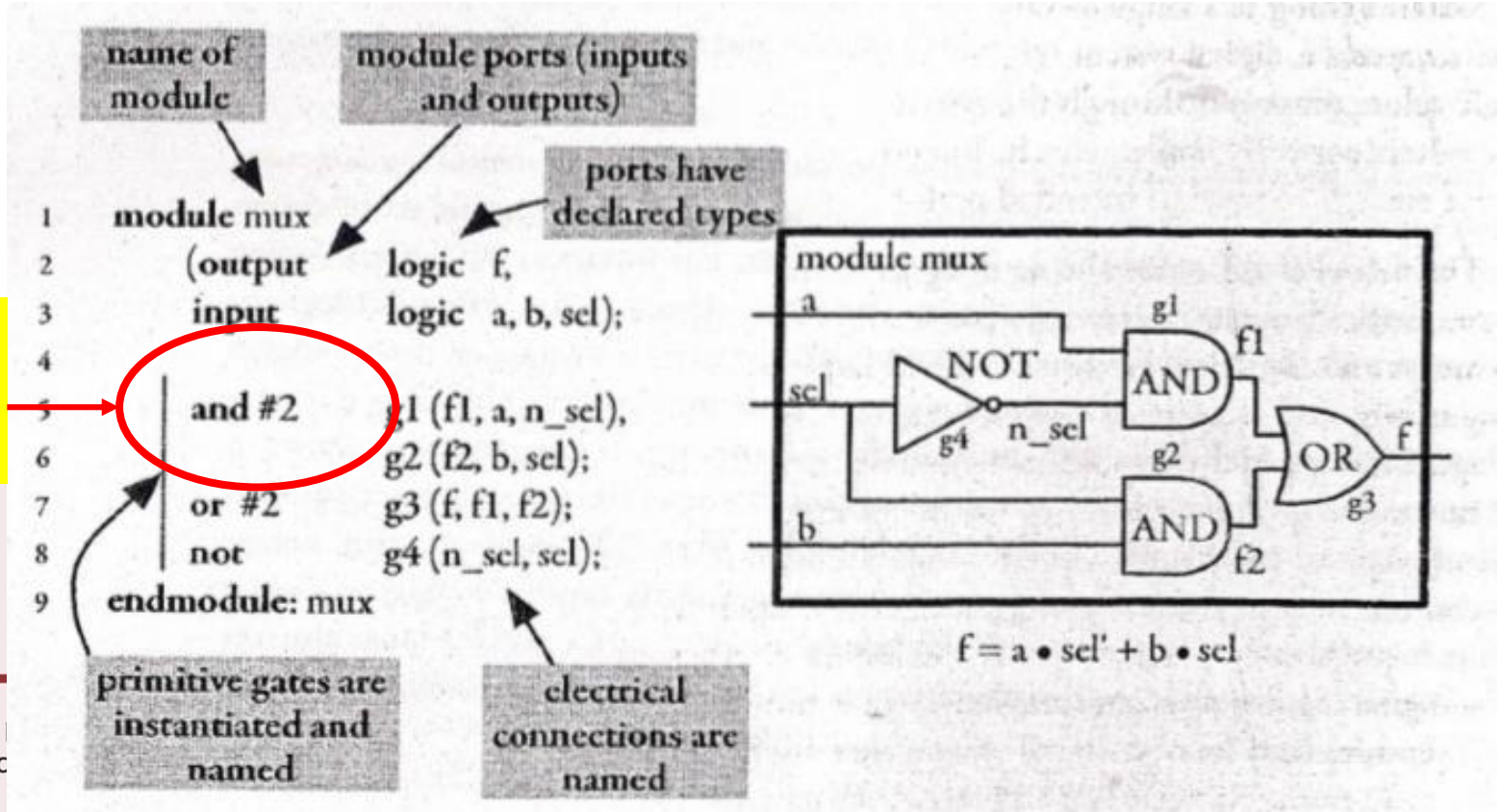
Nível Portas Lógicas – Gate Level

- Module: bloco básico
  - Nome
  - Interface
  - Estrutura interna: conjunto de portas lógicas conectadas



Nível Portas Lógicas – Gate Level

- Portas lógicas: blocos básicos da linguagem
- Conexão por fios
  - Tipo lógico: 0, 1, X. Z (desconectado)
- Retardo (delay) das portas

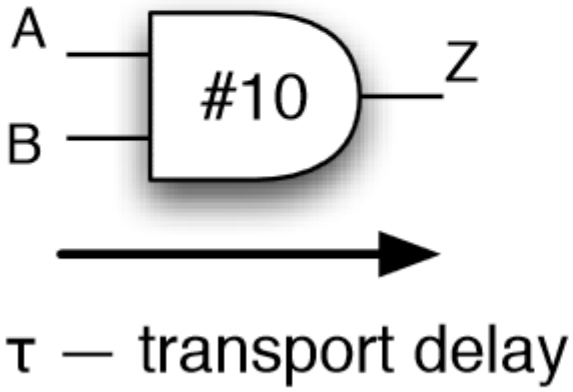


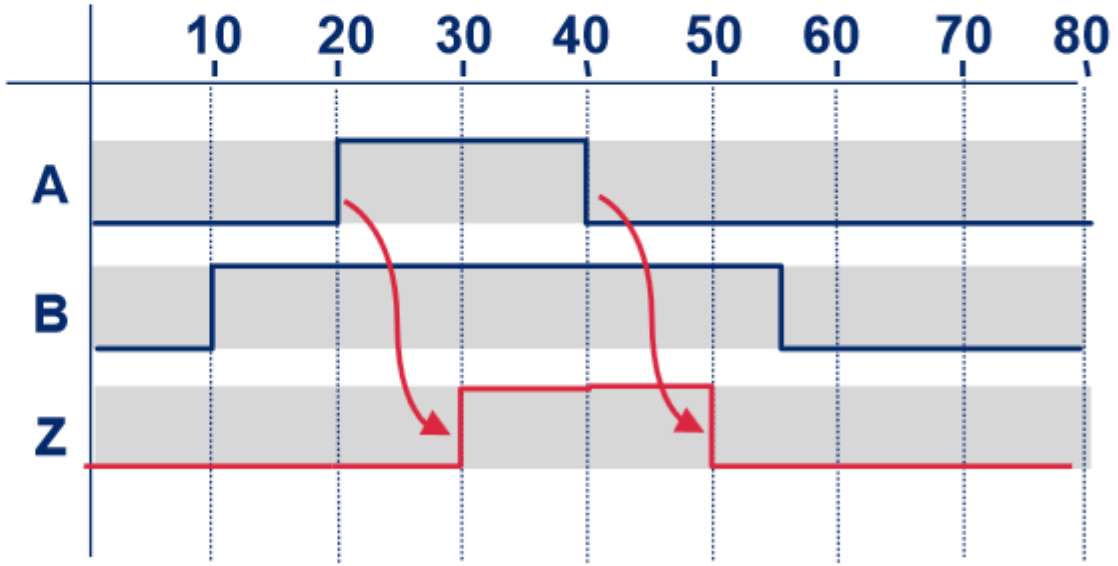
Delay = 2 unidades de tempo

Nível Portas Lógicas – Gate Level

- Retardo (delay) das portas
  - Delay default
  - Diferentes unidades podem ser definidas

- **Example: AND gate**
  - **The output follows, after the specified delay, the inputs according to the AND function**
  - **Delay (#10 here) is the input to output (“transport”) delay**





Simulação do Modelo

- Como simular um modelo?
- Modelo:
  - Conexão de componentes (pode ser em qualquer ordem)
  - Execução do comportamento de cada componente como função das entradas

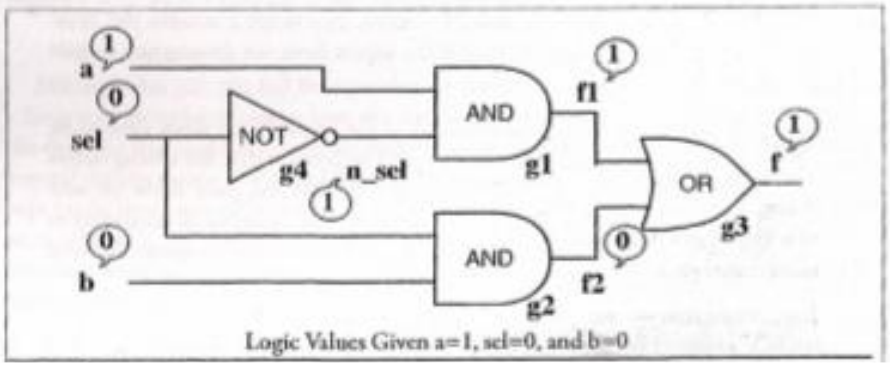
```
1  module mux
2    (output logic  f,
3     input logic  a, b, sel);
4
5     and #2  g1 (f1, a, n_sel),
6             g2 (f2, b, sel);
7     not     g4 (n_sel, sel);
8     or  #2  g3 (f, f1, f2);
9  endmodule: mux
```

```
1  module mux
2    (output logic  f,
3     input logic  a, b, sel);
4
5     or  #2  g3 (f, f1, f2);
6     not     g4 (n_sel, sel);
7     and #2  g1 (f1, a, n_sel),
8             g2 (f2, b, sel);
9  endmodule: mux
```

Example 1.2 — Two Equivalent Modules

Execução do Modelo

- Cada módulo é executável
- Simulador executa os módulos que tiveram os valores em suas portas de entrada alterados
- A alteração na saída é propagada para módulos que estão interconectados considerando o retardo da porta



Execução do Modelo

- Cada módulo é executável
- Simulador executa os módulos que tiveram os valores em suas portas de entrada alterados
- A alteração na saída é propagada para módulos interessados considerando o retardo da porta

Execução em qualquer ordem

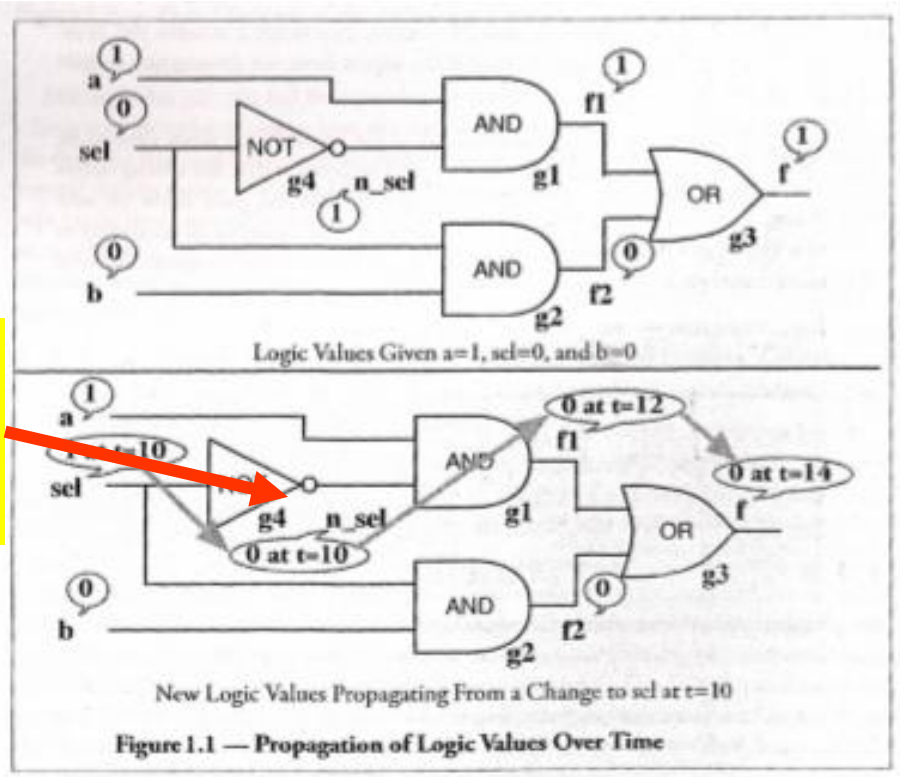


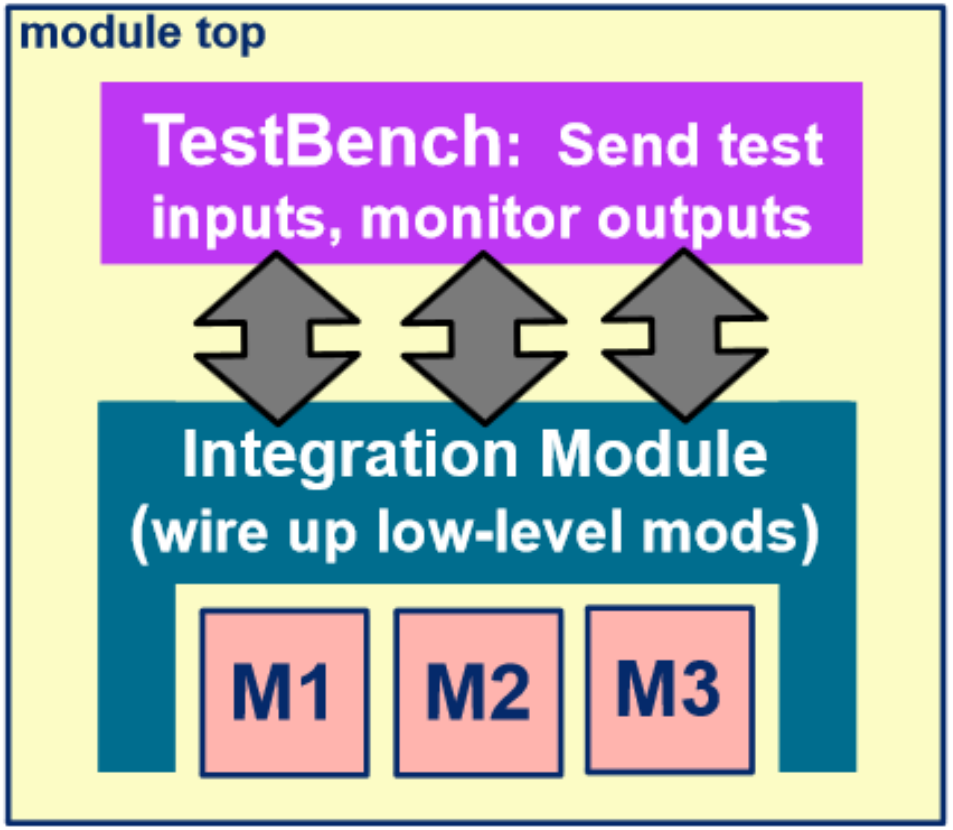
Figure 1.1 — Propagation of Logic Values Over Time

Como Verificar se o projeto está correto?

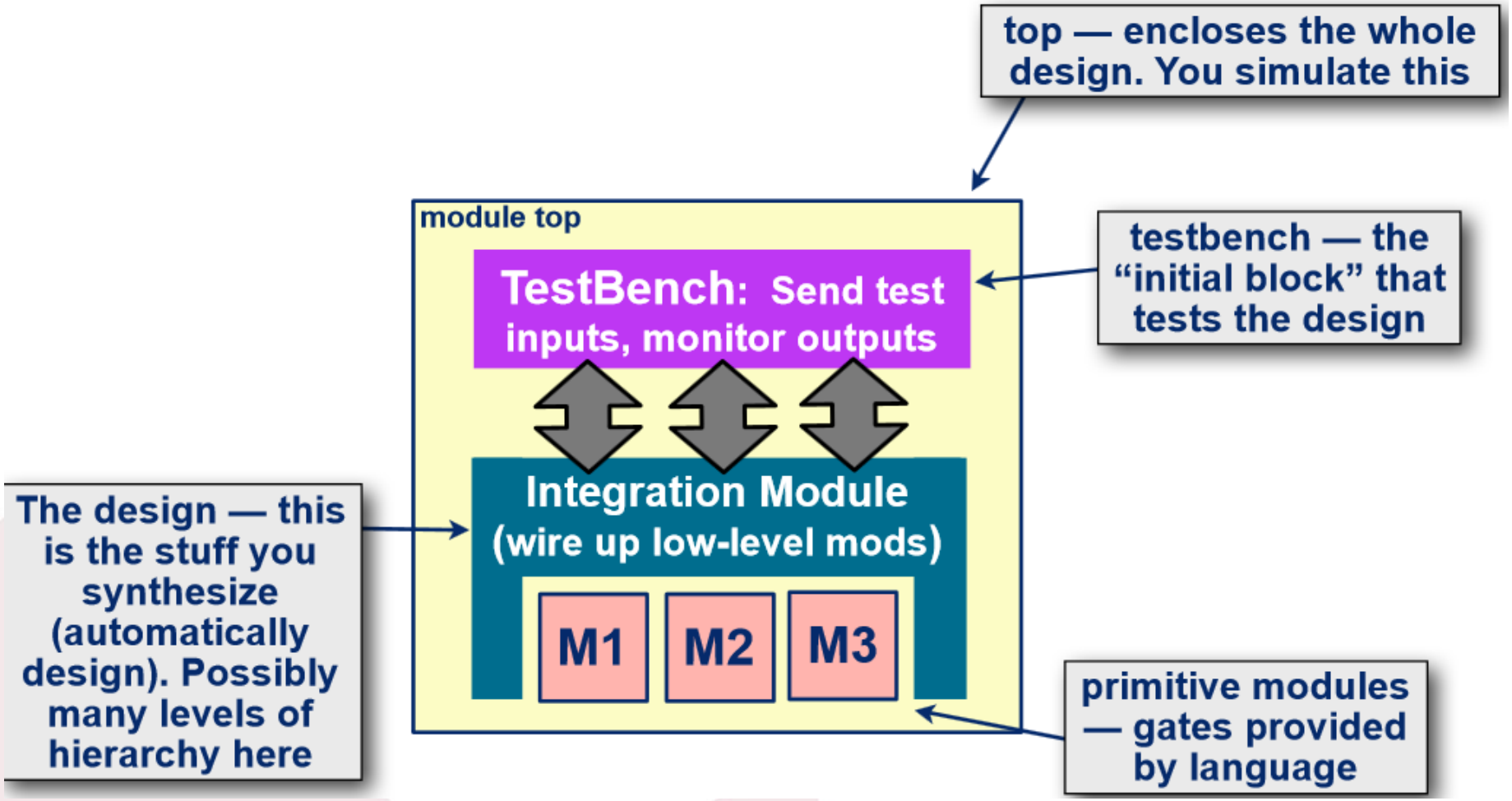
- Modelo do circuito digital
  - Composto de vários módulos
- Gerador de sinais para as entradas
- Visualizador dos sinais nas saídas

Simulando Sistemas Digitais

- Testbench: módulos que vão gerar sinais de entrada para simular um módulo em desenvolvimento e monitorar os sinais de saída do módulo



Simulando Sistemas Digitais



Simulando Sistemas Digitais

- DUT: Device Under Test

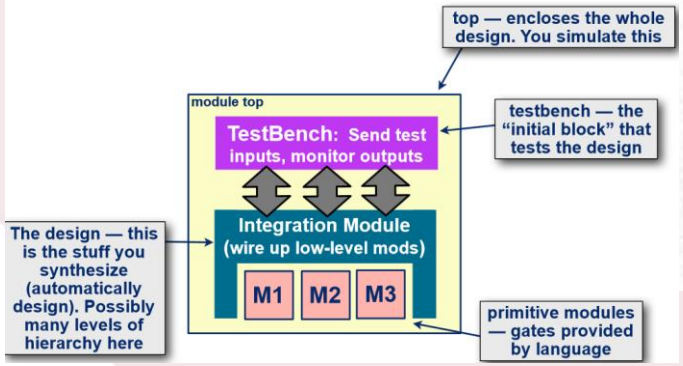
Top

DUT

Testbench

```
1   module muxSim;  //from Example 1.1
2   logic   a, b, sel, n_sel, f1, f2, f;
3
4     and #2   g1 (f1, a, n_sel),
5              g2 (f2, b, sel);
6     or #2    g3 (f, f1, f2);
7     not      g4 (n_sel, sel);
8
9     initial begin
10      $monitor ($time,
11      " a=%b b=%b sel=%b n_sel=%b f1=%b f2=%b f=%b",
12      a, b, sel, n_sel, f1, f2, f);
13      a = 0;
14      b = 0;
15      sel = 0;
16      #12 a = 1;
17      #6 $finish;
18    end
19  endmodule: muxSim
```

Example 1.3 — Module muxSim



Simulando Sistemas Digitais

Centro de Informática U·F·P·E

```
1    0 a=0 b=0 sel=0 n_sel=1 f1=x f2=x f=x
2    2 a=0 b=0 sel=0 n_sel=1 f1=0 f2=0 f=x
3    4 a=0 b=0 sel=0 n_sel=1 f1=0 f2=0 f=0
4   12 a=1 b=0 sel=0 n_sel=1 f1=0 f2=0 f=0
5   14 a=1 b=0 sel=0 n_sel=1 f1=1 f2=0 f=0
6   16 a=1 b=0 sel=0 n_sel=1 f1=1 f2=0 f=1
7   $finish at simulation time 18
```

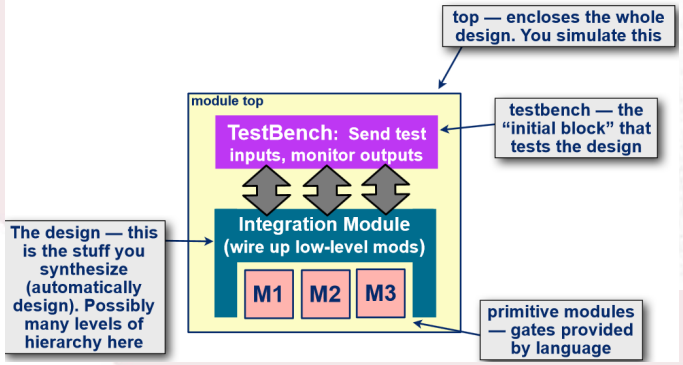
Figure 1.2 — Simulation Results

```
dule muxSim;  //from Example 1.1
ic   a, b, sel, n_sel, f1, f2, f;

nd #2   g1 (f1, a, n_sel),
        g2 (f2, b, sel);
r #2    g3 (f, f1, f2);
ot      g4 (n_sel, sel);
```

Testbench

```
nitial begin
10    $monitor ($time,
11    " a=%b b=%b sel=%b n_sel=%b f1=%b f2=%b f=%b",
12    a, b, sel, n_sel, f1, f2, f);
13    a = 0;
14    b = 0;
15    sel = 0;
16    #12 a = 1;
17    #6 $finish;
18  end
19  endmodule: muxSim
```

Example 1.3 — Module muxSim



Especificando Hierarquia

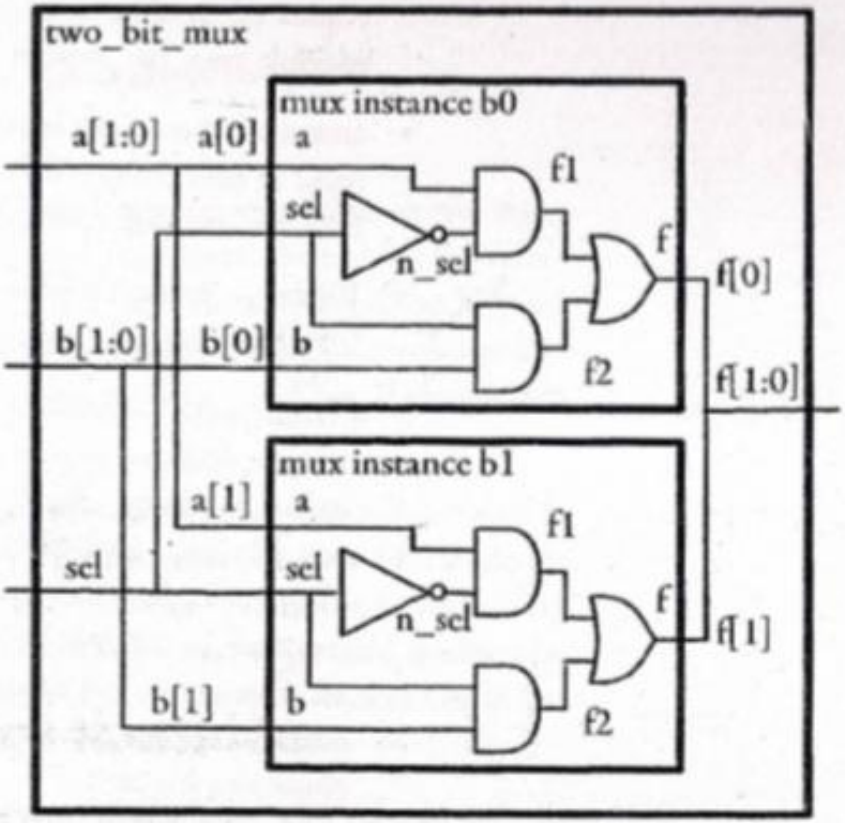


Figure 1.7 — Logic Diagram of two_bit_mux

Especificando Hierarquia

```
1  module two_bit_mux
2     (output  logic   [1:0]  f,
3      input logic     [1:0]  a, b,
4      input logic            sel);
5
6      mux  b0 (f[0], a[0], b[0], sel);
7      mux  b1 (f[1], a[1], b[1], sel);
8  endmodule: two_bit_mux
9
10 module mux
11    (output logic f,
12     input logic   a, b, sel);
13
14     and #2  g1 (f1, a, n_sel),
15             g2 (f2, b, sel);
16     or  #2  g3 (f, f1, f2);
17     not     g4 (n_sel, sel);
18 endmodule: mux
```

Example 1.4 — Module two_bit_mux

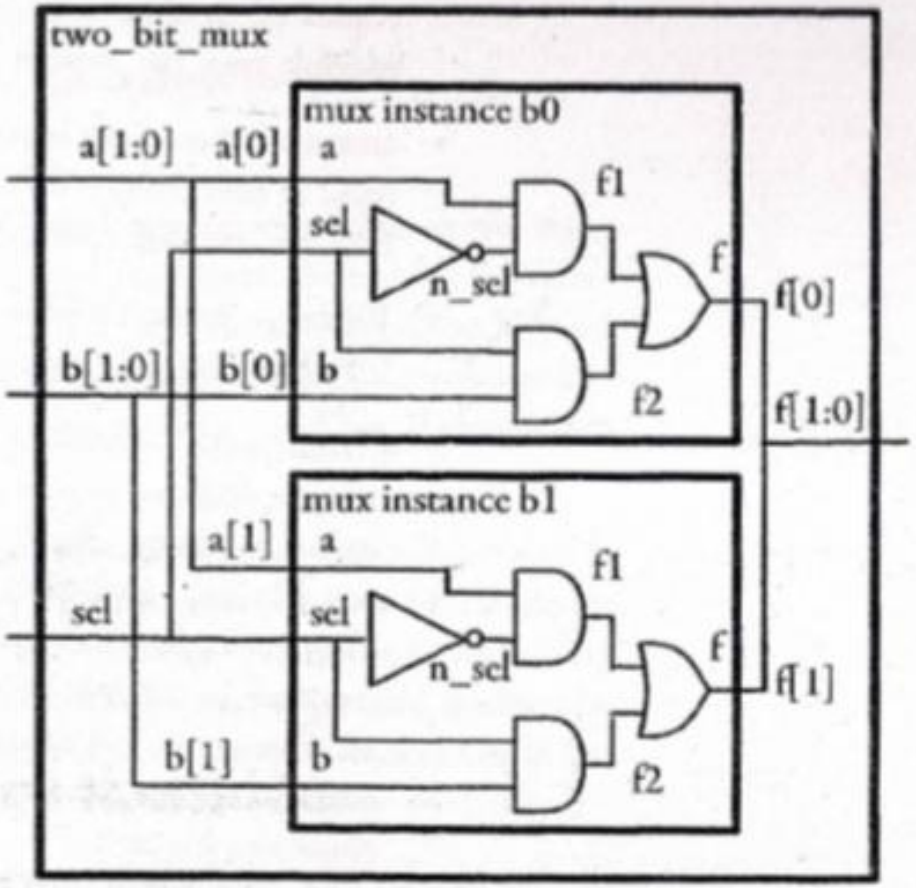


Figure 1.7 — Logic Diagram of two_bit_mux

Mapeamento de Portas

```
1   module mux
2     (output  f,
3      input  m0, m1, sel);
4
5                  .....
6   endmodule: mux
7
```



```
30  endmodule: someNamesMatch
```

Mapeamento de Portas

```
1   module mux
2     (output  f,
3      input m0, m1, sel);
4
5                 .....
6   endmodule: mux
7
8   module orderedPort;
9     logic      sel, in1, in0, result,
10    ...
11    muxa (result, in0, in1, sel);
12  endmodule: orderedPort
13
14  module byNamePort;
15    logic      s, i1, i0, out;
16    ...
17    mux b (.m0(i0), .sel(s), .m1(i1), .f(out));
18  endmodule: byNamePort
19
20  module allNamesMatch
21    logic      m0, m1, sel, f;
22    ...
23    mux c (.*);
24  endmodule: allNamesMatch
25
26  module someNamesMatch
27    logic      m0, m1, select, f;
28    ...
29    mux d (.sel(select), .*);
30  endmodule: someNamesMatch
```

Um Testbench para o Mux

- Como gerar estímulos de forma automática e exaustiva?

Gerador de estímulos

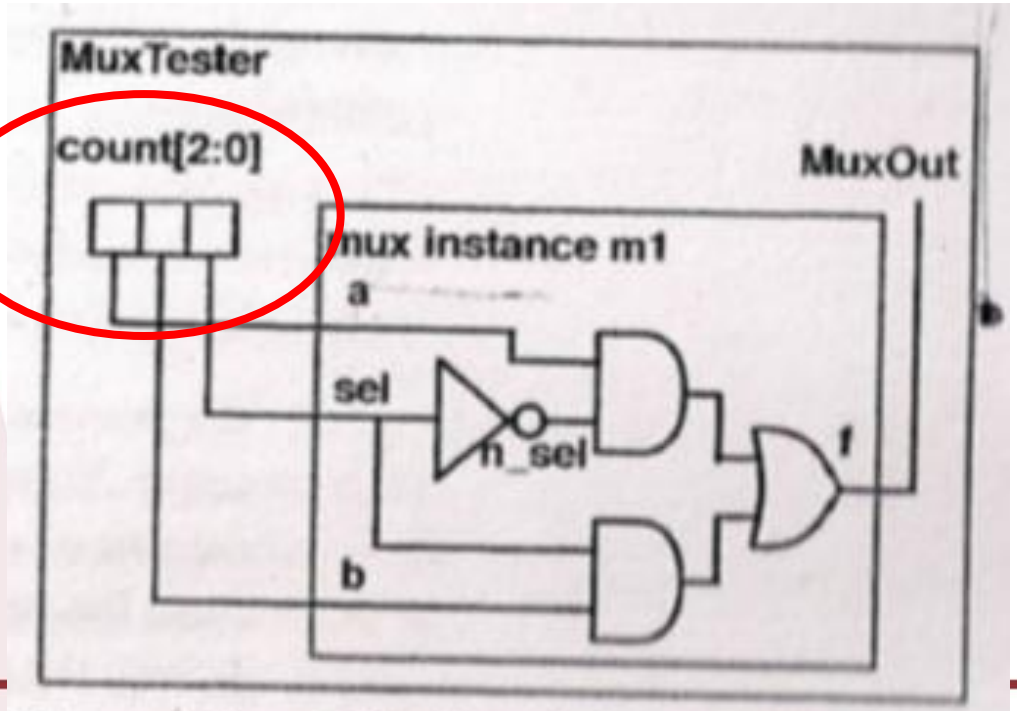


Figure 1.8 — A Testbench for Module mux

Um Testbench para o Mux

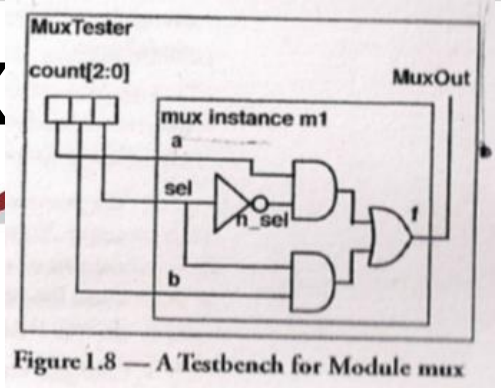


Figure 1.8 — A Testbench for Module mux

```
1   module muxTester;
2     logic  [2:0] count;
3     logic  muxOut;
4
5     mux dut (muxOut, count[2], count[1], count[0]);
6
7     initial begin
8       $monitor ($time,
9       " a b sel = %b, muxOut = %b", count, muxOut);
10
11      for (count = 0; count != 3'b111; count++)
12        #10;
13
14      #10 $finish;
15    end
16  endmodule: muxTester
17
18  module mux
19    (output logic f,
20     input logic  a, b, sel);
21
22    and #2   g1 (f1, a, n_sel),
23             g2 (f2, b, sel);
24    or  #2   g3 (f, f1, f2);
25    not      g4 (n_sel, sel);
26  endmodule: mux // printout is below
```

Gerador
de estímulos

Um Testbench para o Mux

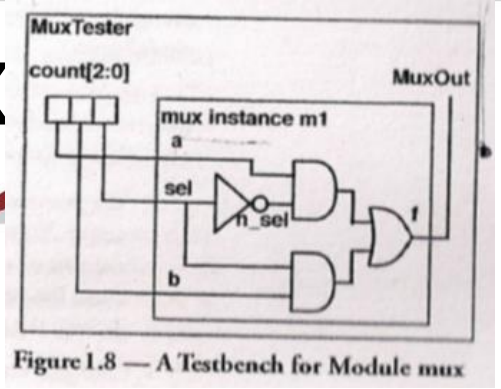


Figure 1.8 — A Testbench for Module mux

```
1   module muxTester;
2     logic  [2:0] count;
3     logic  muxOut;
4
5     mux dut (muxOut, count[2], count[1], count[0]);
6
7     initial begin
8       $monitor ($time,
9       " a b sel = %b, muxOut = %b", count, muxOut);
10
11      for (count = 0; count != 3'b111; count++)
12        #10;
13
14      #10 $finish;
15    end
16  endmodule: muxTester
17
18  module mux
19    (output logic f,
20     input logic  a, b, sel);
21
22    and #2   g1 (f1, a, n_sel),
23             g2 (f2, b, sel);
24    or  #2   g3 (f, f1, f2);
25    not      g4 (n_sel, sel);
26  endmodule: mux // printout is below
```

Gerador de estímulos

```
endmodule: mux // printout is below
                0 a b sel = 000, muxOut = x
                4 a b sel = 000, muxOut = 0
               10 a b sel = 001, muxOut = 0
               20 a b sel = 010, muxOut = 0
               30 a b sel = 011, muxOut = 0
               34 a b sel = 011, muxOut = 1
               40 a b sel = 100, muxOut = 1
               50 a b sel = 101, muxOut = 1
               54 a b sel = 101, muxOut = 0
               60 a b sel = 110, muxOut = 0
               64 a b sel = 110, muxOut = 1
               70 a b sel = 111, muxOut = 1
$finish at simulation time                80
```

Um testbench mais inteligente

```
1  module muxTester;
2    logic  [2:0] count;
3    logic  muxOut;
4
5    mux dut (muxOut, count[2], count[1], count[0]);
6
7    initial begin
8      $monitor ($time,
9      " a b sel = %b, muxOut = %b", count, muxOut);
10
11     for (count = 0; count != 3'b111; count++)
12       #10;
13
14     #10 $finish;
15   end
16 endmodule: muxTester
17
18 module mux
19   (output logic f,
20    input logic  a, b, sel);
21
22   and #2  g1 (f1, a, n_sel),
23           g2 (f2, b, sel);
24   or  #2  g3 (f, f1, f2);
25   not     g4 (n_sel, sel);
26 endmodule: mux // printout is below
```

Como saber se a especificação está correta?

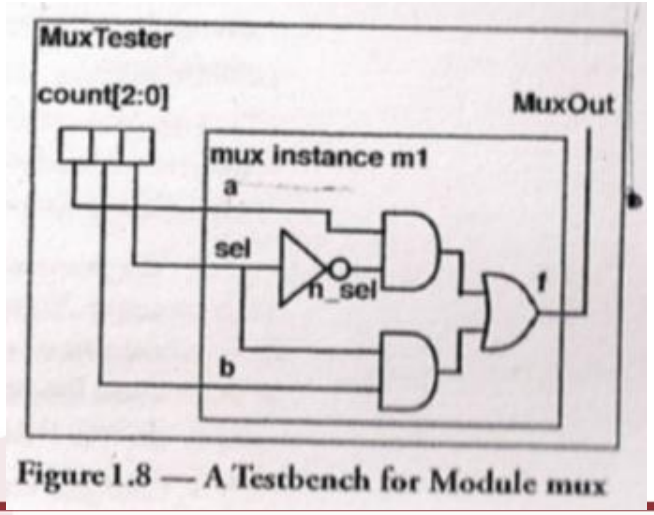


Figure 1.8 — A Testbench for Module mux

Um testbench mais inteligente

```
1  module muxTester;
2    logic  [2:0] count;
3    logic  muxOut;
4
5    mux dut (muxOut, count[2], count[1], count[0]);
6
7    initial begin
8      $monitor ($time,
9      " a b sel = %b, muxOut = %b", count, muxOut);
10
11
12
13
14     #10 $finish;
15   end
16 endmodule: muxTester
17
18 module mux
19   (output logic f,
20    input logic  a, b, sel);
21
22   and #2  g1 (f1, a, n_sel),
23           g2 (f2, b, sel);
24   or  #2  g3 (f, f1, f2);
25   not     g4 (n_sel, sel);
26 endmodule: mux // printout is below
```

```
1 for (count = 0; count != 3'b111; count++) begin
2   #10;
3   if (count[0])      // if sel is TRUE
4     if (muxOut != count[1])// if muxOut is != b
5       $display("oops: a b sel = %b, muxOut =
    %b", count, muxOut);
6     else if (muxOut != count[2]) // if muxOut is != a
7       $display("oops: a b sel = %b, muxOut =
    %b", count, muxOut);
8 end
```

Compara com valor de referência

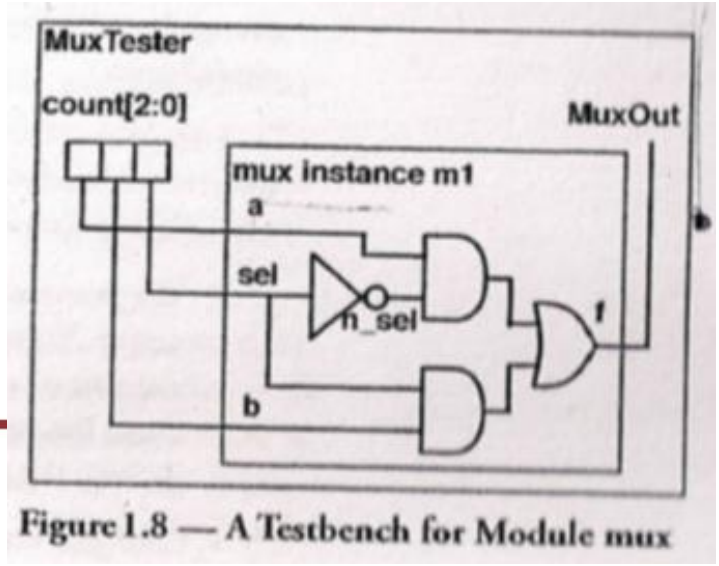


Figure 1.8 — A Testbench for Module mux

Resumo

- Foram apresentados os principais conceitos de simulação baseada em eventos
- Tempo e ação concorrente
  - Modelagem do tempo de simulação
  - Modelagem de componentes que são ativados concorrentemente
- Módulos, Instanciação e Hierarquia
  - Modelos de partes (componentes de hardware)
  - Projeto Bottom-Up
- Modelos estruturais e Procedurais
  - Estrutura: interconexão de componentes
  - Procedural: Comportamento do Testbench

Projetando Circuitos Combinacionais

Circuitos Combinacionais

- Lógica combinacional:
  - Saída: função booleana das entradas
    - F: I -> O
    - Retardo de propagação: tempo para o valor da saída ficar estável em função de mudança nos valores de entrada.

Circuitos Combinacionais

- Especificação em SystemVerilog

Always_comb

```
1   module sum_and_dif_A
2     (output  logic [3:0]  result,
3      input   logic [3:0]  a, b,
4      input                select_plus);
5
6     always_comb
7       if (select_plus)
8         result = a + b;
9       else result = a - b;
10  endmodule: sum_and_dif_A
```

comportamemto

Assign

```
12  module sum_and_dif_B
13    (output  logic [3:0]  result,
14     input   logic [3:0]  a, b,
15     input   logic        select_plus);
16
17    assign result = (select_plus) ? a +b : a - b;
18  endmodule: sum_and_dif_B
```

comportamento

Example 2.1 — always_comb and assign

Circuitos Combinacionais

- Especificação em SystemVerilog

Assign múltiplas saídas

```
1   module compare
2     (output  logic        eq, neq,
3      input   logic [3:0]  value);
4
5     assign  neq = ~eq,
6             eq = (value == 0);
7   endmodule: compare
```

Example 2.2 — Multiple assign Statements

Circuitos Combinacionais

- Especificação em SystemVerilog

Hierarquia

```
1  module add_sub_compare
2    (output logic [3:0] result,
3     input  logic [3:0] a, b,
4     output logic       neq, eq,
5     input  logic       plus_minus);
6
7    sum_and_dif_A alu (result, a, b, plus_minus);
8    compare       c (eq, neq, result);
9  endmodule: add_sub_compare
```

Example 2.3 — Module Instantiation

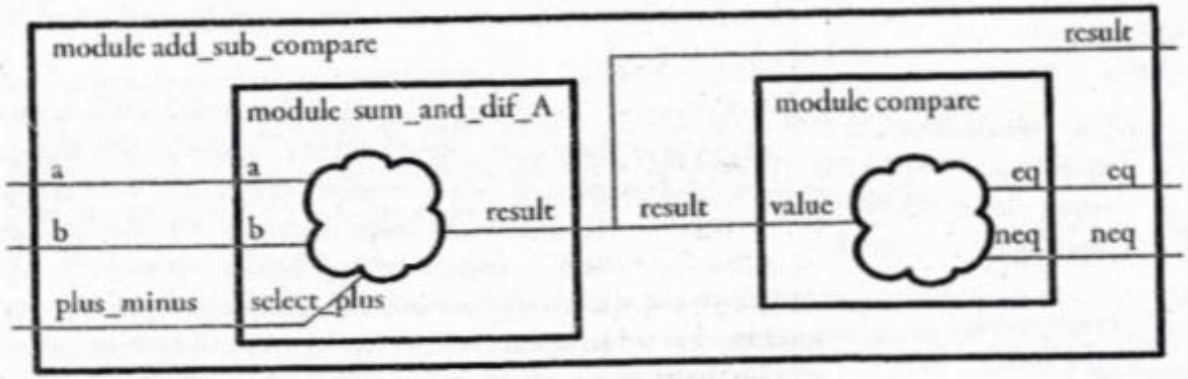


Figure 2.1 — Module Instantiation and Naming

Circuitos Combinacionais

- Porque o circuito abaixo não é combinacional?

```
1   module notCombinational
2     (input logic  [3:0]  a, b,
3      output logic  [3:0]  sum,
4      input logic          hold);
5
6     always_comb
7       if (~hold)
8         sum = a + b;
9   endmodule: notCombinational
```

Necessidade de latchs para guardar o valor

A saída não está especificada para todas as entradas

Circuitos Combinacionais

- Formalizando....
- **always_comb** statement
  - always_comb sum=b+c
  - always_comb begin

    sum=b+c;

    dif = b-c;

    end
- Executa repetidas vezes enquanto valores mudam

Circuitos Combinacionais

- Formalizando....
- **assign** statement
  - assign sum=b+c
  - assign sum=b+c, dif = b-c;

- Cada expressão (lado direito) é avaliada continuamente

Circuitos Combinacionais

- Construtores de linguagens de programação
- **If, if-else, case**
- f = (a . b) + (b . c) + (a . c)
- assign f = (a & b) | (b & c) | (a & c)

```
1   module if1
2     (input   logic  a, b, c,
3      output  logic  f);
4
5     always_comb begin
6       f = 0;
7       if (a & b)   f = 1;
8       if (c & (a ^ b)) f = 1;
9     end
10  endmodule: if1
```

```
12  module if2
13    (input   logic  a, b, c,
14     output  logic  f);
15
16    always_comb begin
17      if (a & b)   f = 1;
18      else if (c & a ^ b)
19        f = 1;
20      else f = 0;
21    end
22  endmodule: if2
```

Circuitos Combinacionais

- Construtores de linguagens de programação
- **If, if-else, case**
- f = (a . b) + (b . c) + (a . c)

Concatenação

Constante com tam. Pré definido

```
1   module basicCase
2     (input logic   a, b, c,
3      output  logic   f);
4
5     always_comb
6       case ({a, b, c})
7         3'b000: f = 0;
8         3'b001: f = 0;
9         3'b010: f = 0;
10        3'b011: f = 1;
11        3'b100: f = 0;
12        3'b101: f = 1;
13        3'b110: f = 1;
14        3'b111: f = 1;
15      endcase
16  endmodule: basicCase
```

Circuitos Combinacionais

- Construtores de linguagens de programação
- **If, if-else, case**
- f = (a . b) + (b . c) + (a . c)

```
always_comb
  case ({a, b, c})
    3'b000: f = 0;
    3'b001: f = 0;
    3'b010: f = 0;
    3'b100: f = 0;
    default: f = 1; // use default
  endcase
```

Valores default

Circuitos Combinacionais

- Construtores de linguagens de programação
- **If, if-else, case**
- f = (a . b) + (b . c) + (a . c)

```
1    always_comb
2      case ({a, b, c})
3        3'b000,
4        3'b001: f = 0; // execute this statement if case expression is
                                                    3'b000 or 3'b001
5        3'b010: f = 0;
6        3'b100: f = 0;
7        default: f = 1;
8      endcase
```

Facilitando comparação

Especificando Testbenches

- **Papel do testbench:**
  - **Modelar ambiente**
  - **Depurar o projeto**
- **Não é sintetizável**
- **Visualização dos sinais**
  - **$ monitor**
  - **S display**
  - **$ strobe**

Especificando Testbenches

Table 2.1 — Printing Statements for Testbenches

Statement type	When it prints	What it's used for
$monitor (...)	A monitor is a concurrently acting statement that prints anytime one of its inputs changes. The values printed are those existing at the end of the current simulation time. Thus you are guaranteed that they will be consistent with the end of the time listed in its printout.	A monitor provides a snapshot of values at the end of a simulation time. There can be only one monitor active. Calling monitor with a different set of inputs cancels the previous monitor and activates the new one.

Especificando Testbenches

Centro de Informática

Table 2.1 — Printing Statements for Testbenches		
Statement type	**When it prints**	**What it's used for**
$display (…)	display is like a print statement in a programming language. It is a procedural statement that is executed when the model (e.g. always_comb block) it is found in executes. The values existing when it is called are printed. Note that the values printed may be different than what a monitor would print for the same variable. Also note that combinational blocks may execute several times during a time period due to glitches (hazards) on their inputs; a display statement in them would then print several times too, possibly with different values.	A display is used to provide information about a model as it is executing. It can give you an idea if a certain part of the code is being reached and what the values are at that point.
$strobe (…)	strobe differs from display only by when it prints. It prints at the end of the current time using the values existing then.	A strobe is used like a display, to provide information about the model as it is executing.

Exemplo Testbench Básico

```
1   module if2
2     (input   logic a, b, c,
3      output  logic f);
4
5     always_comb begin
6       if (a & b)   f = 1;
7       else  if (c & a ^ b)
8               f = 1;
9             else f = 0;
10    end
```

```
12
13  module top;
14    logic  [2:0]  count;
15    logic         result;
16
17    if2  dut (count[2], count[1], count[0], result);
18
19    initial begin
20      $monitor ($time, " abc=%b, f=%b",
21                count, result);
22      for (count = 0; count != 3'b111; count++)
23        #1;
24      #1 $finish;
25    end
26  endmodule: top
```

Instancia do DUV

testbench

Resultado

```
1   0 abc=000, f=0
2   1 abc=001, f=0
3   2 abc=010, f=1
4   3 abc=011, f=1
5   4 abc=100, f=0
6   5 abc=101, f=1
7   6 abc=110, f=1
8   7 abc=111, f=1
```

Figure 2.3 — Simulation Results of Example 2.13

Exemplo Testbench Básico

```
1  module fourBitAdder
2     (input   logic  [3:0]  a, b,
3      output  logic  [3:0]  sum,
4      input   logic         cIn,
5      output  logic         cOut);
6     logic    [2:0]         c;
7
8     adder    b0 (a[0], b[0], cIn, sum[0], c[0]);
9     adder    b1 (a[1], b[1], c[0], sum[1], c[1]);
10    adder    b2 (a[2], b[2], c[1], sum[2], c[2]);
11    adder    b3 (a[3], b[3], c[2], sum[3], cOut);
12 endmodule: fourBitAdder
13
14 module adder
15    (input logic      a, b, cI,
16     output  logic    s, cO);
17
18    assign  s = a ^ b ^ cI,
19            cO = (a&b) | (a&cI) | (b&cI);
20 endmodule: adder
21
```

Adicionador 4 bits

Adicionador 1 bit

Exemplo Testbench Básico

```
1   module fourBitAdder
2     (input   logic  [3:0]  a, b,
3      output  logic  [3:0]  sum,
4      input   logic         cIn,
5      output  logic         cOut);
6     logic    [2:0]         c;
7
8     adder    b0 (a[0], b[0], cIn, sum[0], c[0]);
9     adder    b1 (a[1], b[1], c[0], sum[1], c[1]);
10    adder    b2 (a[2], b[2], c[1], sum[2], c[2]);
11    adder    b3 (a[3], b[3], c[2], sum[3], cOut);
12  endmodule: fourBitAdder
13
14  module adder
15    (input logic      a, b, cI,
16     output  logic    s, cO);
17
18    assign  s = a ^ b ^ cI,
19            cO = (a&b) | (a&cI) | (b&cI);
20  endmodule: adder
21
```

```
22  module test;
23    logic  [3:0]  s;
24    logic         cOut;
25    logic  [9:0]  count;
26
27    fourBitAdder
28        add0 (count[7:4], count[3:0], s, count[8], cOut);
29
30    initial begin
31      for (count = 0; count <= 10'h200; count++)
32        #1 if ({cOut, s} != (count[7:4] + count[3:0] + count[8]))
33          $display ("oops! %d != %d + %d + %d",
34            {cOut, s}, count[7:4], count[3:0], count[8]);
35      $finish;
36    end
37  endmodule: test
```

Tipos de Dados - Principais

Centro de Informática U·F·P·E

Table 2.2 — Integral Data Types					
type name	2 or 4 state	size (bits)	signed/ unsigned default	value at simulation startup	other
shortint	2	16	signed	0	Same as short in C.
int	2	32	signed	0	Same as int in C.
longint	2	64	signed	0	Same as long in C.
bit	2	user-defined	unsigned	0	
byte	2	8	signed or ascii character	0	Same as "signed bit [7:0] varName;"

Tipos de Dados - Principais

Centro de Informática U·F·P·E

Table 2.2 — Integral Data Types					
type name	2 or 4 state	size (bits)	signed/ unsigned default	value at simulation startup	other
logic	4	user-defined	unsigned	x	
reg	4	user-defined	unsigned	x	This was a Verilog type; it has been superseded by the logic type in SystemVerilog.
integer	4	32	signed	x	Not the same as int in C because this is 4-state
time	4	64	unsigned only	0	Not used in modeling a design, but can be used in testbenches.

Tipos de Dados - Principais

Centro de Informática U·F·P·E

Table 2.2 — Integral Data Types					
type name	**2 or 4 state**	**size (bits)**	**signed/ unsigned default**	**value at simulation startup**	**other**
shortint	2	16	signed	0	Same as short in C.
int	2	32	signed	0	Same as int in C.
longint	2	64	signed	0	Same as long in C.
bit	2	user-defined	unsigned	0	
byte	2	8	signed or ascii character	0	Same as "signed bit [7:0] varName;"
logic	4	user-defined	unsigned	x	
reg	4	user-defined	unsigned	x	This was a Verilog type; it has been superseded by the logic type in SystemVerilog.
integer	4	32	signed	x	Not the same as int in C because this is 4-state
time	4	64	unsigned only	0	Not used in modeling a design, but can be used in testbenches.

Vetores de Logics

- Para a definição de vetores de bits procedemos das formas apresentadas abaixo.
  - Vetor unidimensional

    logic [msb: lsb] nome_vetor;
  - Vetor bidimensional

    logic [msb: lsb] nome_vetor [minimo :máximo];
  - Vetor tridimensional

    logic [msb:lsb] [máximo:mínimo] nome_vetor[mínimo: máximo]

Enumeration

- Maneira de usar constantes de forma controlada

```
1   module datapath_enum;
2     enum logic [2:0] {ADD, SUB, AND, OR, XOR} op;
3     ...
4     always_comb
5       case (op)
6         ADD: result = a + b;
7         SUB:  result = a - b;
8         AND: result = a & b;
9         OR:   result = a | b;
10        XOR: result = a ^ b;
11        default:  result = 8'bxxxx_xxxx;
12      endcase
13  endmodule: datapath_enum
```

Enumeration

- Maneira de usar constantes de forma controlada

```
1   enum logic [2:0]
2     {ADD=  3'b100,
3      SUB=  3'b010,
4      AND=  3'b001,
5      OR=   3'b110,
6      XOR=  3'b011} op;
```

```
1   module datapath_enum;
2     enum logic [2:0] {ADD, SUB, AND, OR, XOR} op;
3     ...
4     always_comb
5       case (op)
6         ADD: result = a + b;
7         SUB: result = a - b;
8         AND: result = a & b;
9         OR:  result = a | b;
10        XOR: result = a ^ b;
11        default: result = 8'bxxxx_xxxx;
12      endcase
13  endmodule: datapath_enum
```

Resumo

- Especificação RTL e Simulação de Sistemas Digitais
- Modelando Circuitos Combinacionais
  - Assign
  - Always_comb
    - Construtores procedurais
- Tipos de Dados

Projetando Circuitos Sequenciais

Circuitos Sequenciais

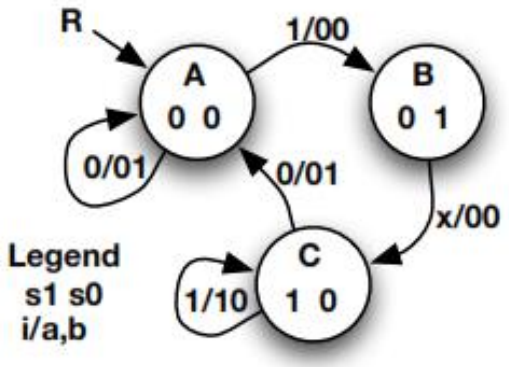
- O que é um elemento sequencial?
  - Um flip flop
  - Um latch
- São usados para armazenar informação do Sistema:
  - O estado do Sistema

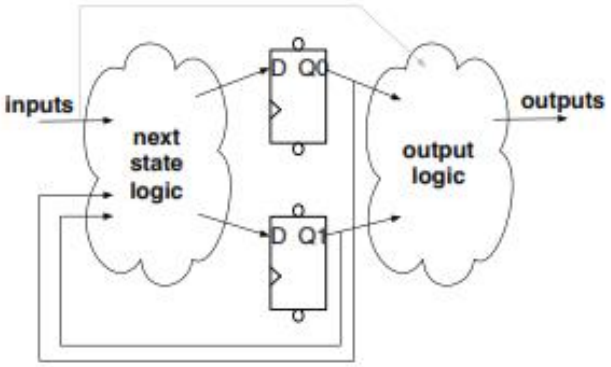
Circuitos Sequenciais

- Como especificar elementos sequenciais em uma descrição RTL?
- Elementos sequenciais não são especificados ***Explicitamente***
- Eles são ***inferidos*** a partir de como é feita a sua especificação

Circuitos Sequenciais

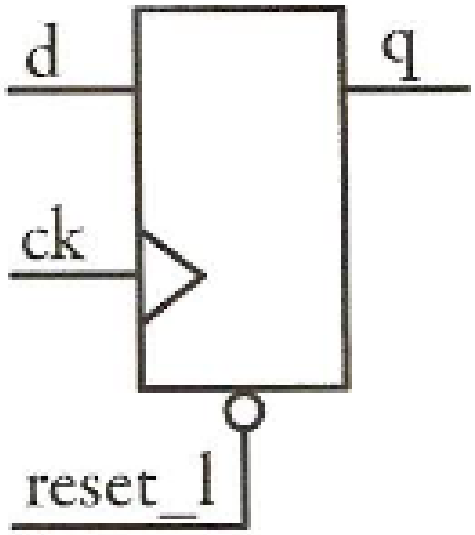
- Circuitos sequenciais juntamente com circuitos combinacionais permitem a implementação de uma máquina de estados FSM
  - Sinais de sincronização
    - Clock
    - Reset
  - Estados
  - Entradas e saídas





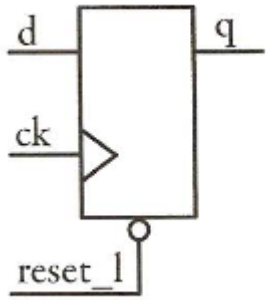
Flip-Flop tipo D

- Flip-flop tipo D
  - Armazenam 1 bit
  - Copia a entrada na transição positiva do clock
  - Reseta a saída na transição negativa do reset



Especificando um Flip-Flop

- Flip-flops que são edge-triggered
  - Informado pelo uso do simbolo @ com a palavra posedge ou negedge
- Efeito:
  - Todas as varíaveis que estão no lado esquerdo da atribuíção com “<=” serão implementadas com Flip-flops trigados na transição do clock ou reset.
  - Reset é assíncrono
  - <= atribuição concorrente



Modelo Flip-flop: loop contínuo – mudanças nos sinais após @

```
module Dff
   (output  logic Q,
    input         clk, d, resetN);

   always_ff @(posedge clk,
               negedge resetN)
      if (~resetN)
         Q <= 0;
      else
         Q <= d;
endmodule
```

Especificando um Registrador

- Registrador:
  - Concatenação de vários flip-flops

```
1   module reg8
2    (output  logic  [7:0]  q,
3     input   logic  [7:0]  d,
4     input   logic         ck, reset_l);
5
6     always_ff @(posedge ck, negedge reset_l)
7       if (~reset_l)
8             q <= 0;
9       else  q <= d;
10  endmodule: reg8
```

Example 3.2 — An 8-Bit Register

Entradas e saídas de 8 bits

Tamanho parametrizado

```
1   module register
2     #(parameter  W = 8)
3     (output  logic  [W-1:0]  q,
4      input   logic  [W-1:0]  d,
5      input   logic          ck, reset_l);
6
7     always_ff @(posedge ck, negedge reset_l)
8       if (~reset_l)
9             q <= 0;
10      else  q <= d;
11  endmodule: register
```

Example 3.3 — Parameterized Register

Instanciação

```
ule datapath;
3     register  #(32)  regA (Q, D, clock, r_l);
4   endmodule: datapath
```

Example 3.4 — Instantiation With Parameter

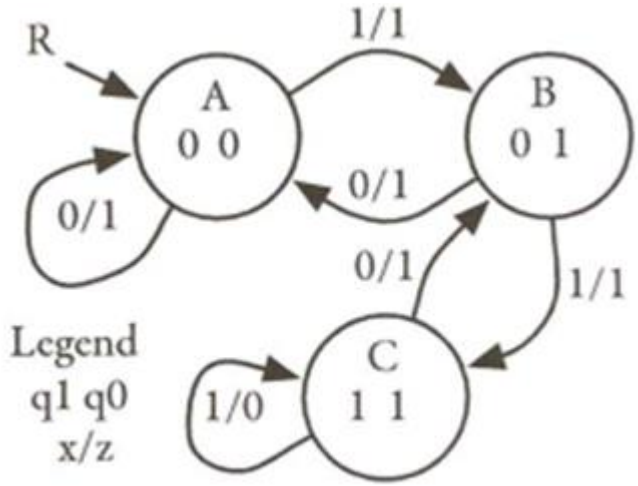
Máquina de Estados Finitos FSM

- Definidas formalmente como um conjunto de estados, reset e clock.
- Conjunto de combinações das entradas.
  - Não necessariamente todas as 2^n são possíveis devido aos don't-cares
- Combinações das saídas
  - Não necessariamente todas as 2^n serão possíveis
- Função do próximo estado (δ) e função da saída (λ) são combinacionais
- Clock
- Sinal reset

Máquina de Estados Finitos FSM

Modelo de Computação:

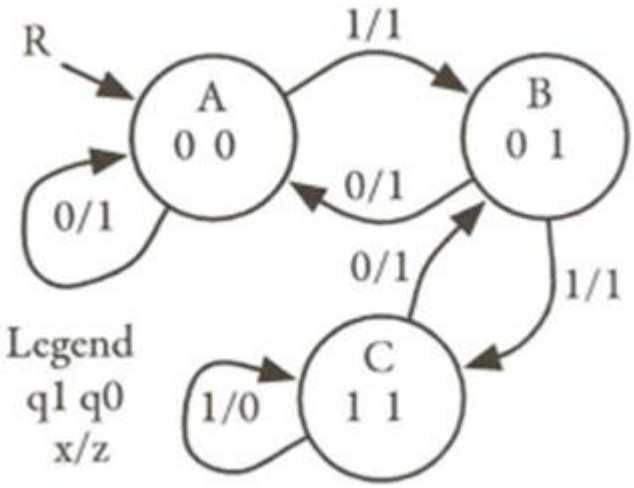
iniciando no estado de reset, uma transição positiva do clock causa o Sistema mudar para outro (ou mesmo) estado como definido pela função δ

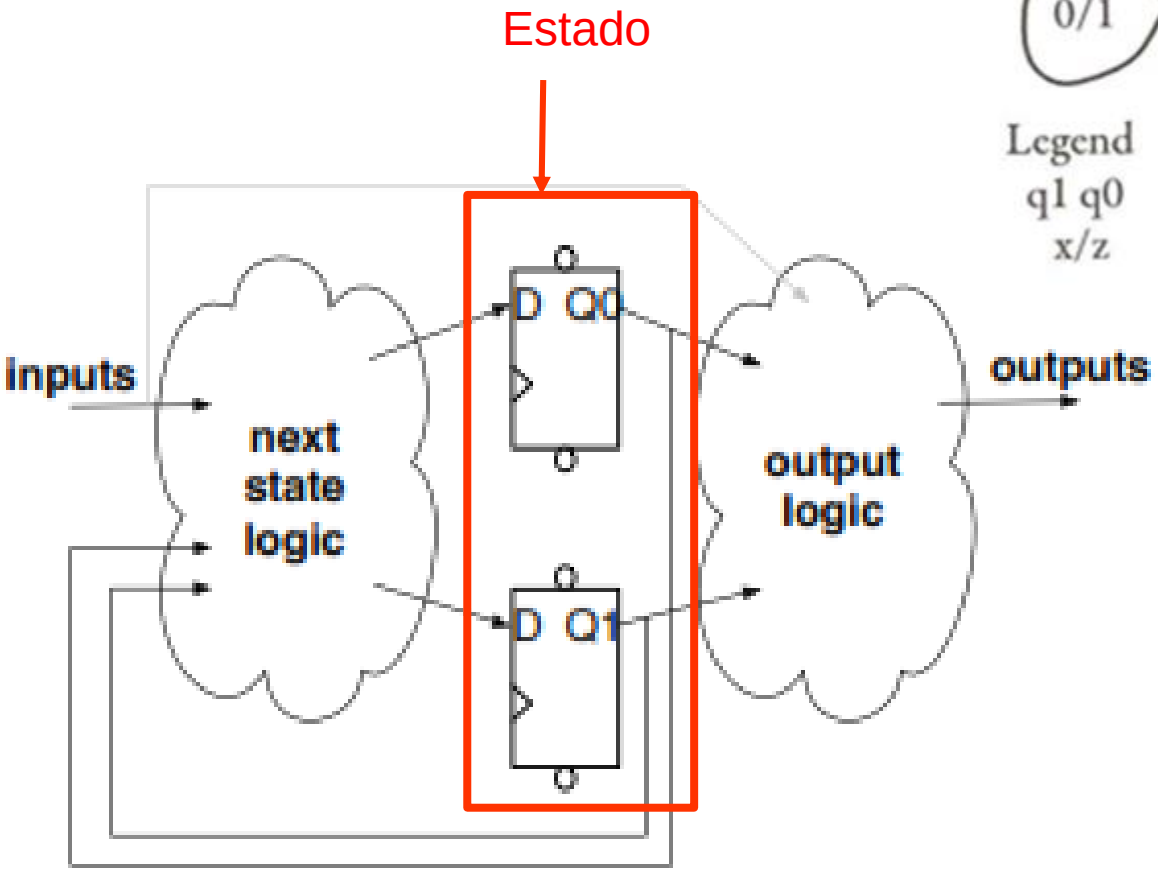


Máquina de Estados Finitos FSM

Centro de Informática U·F·P·E

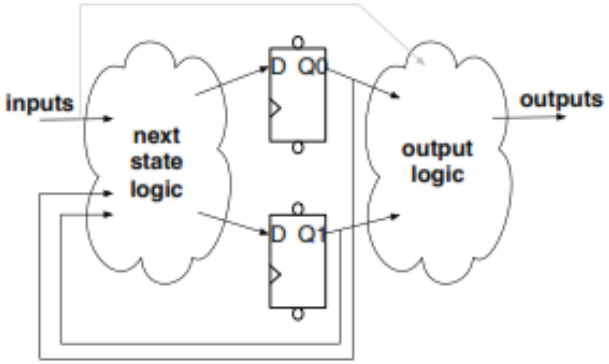
Modelo de Implementação

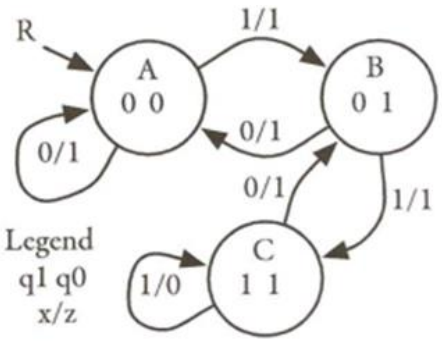




FSM em SystemVerilog

Modelo de Implementação





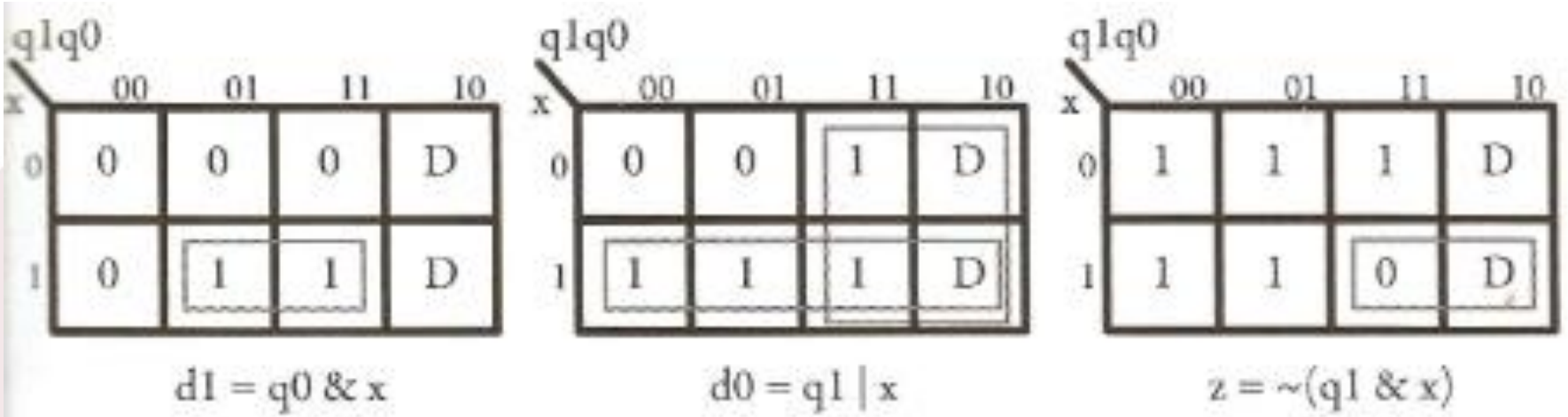


Figure 3.2 — The Karnough Maps

FSM em SystemVerilog

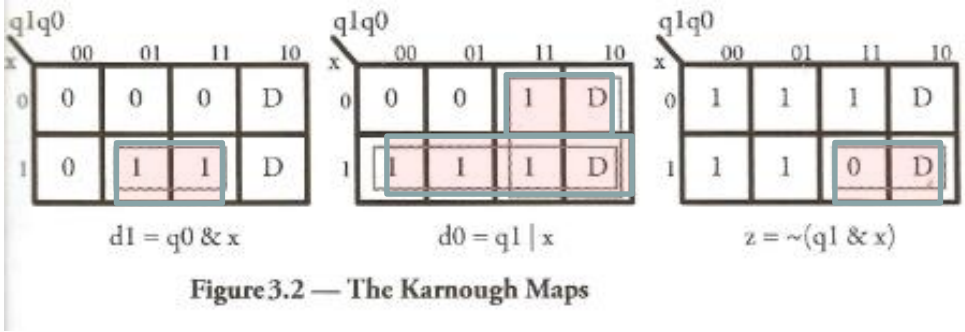
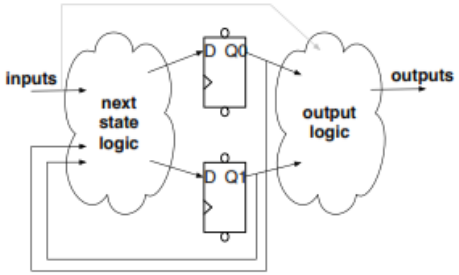
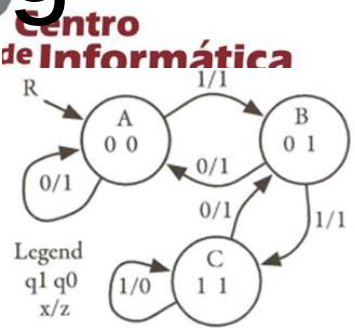
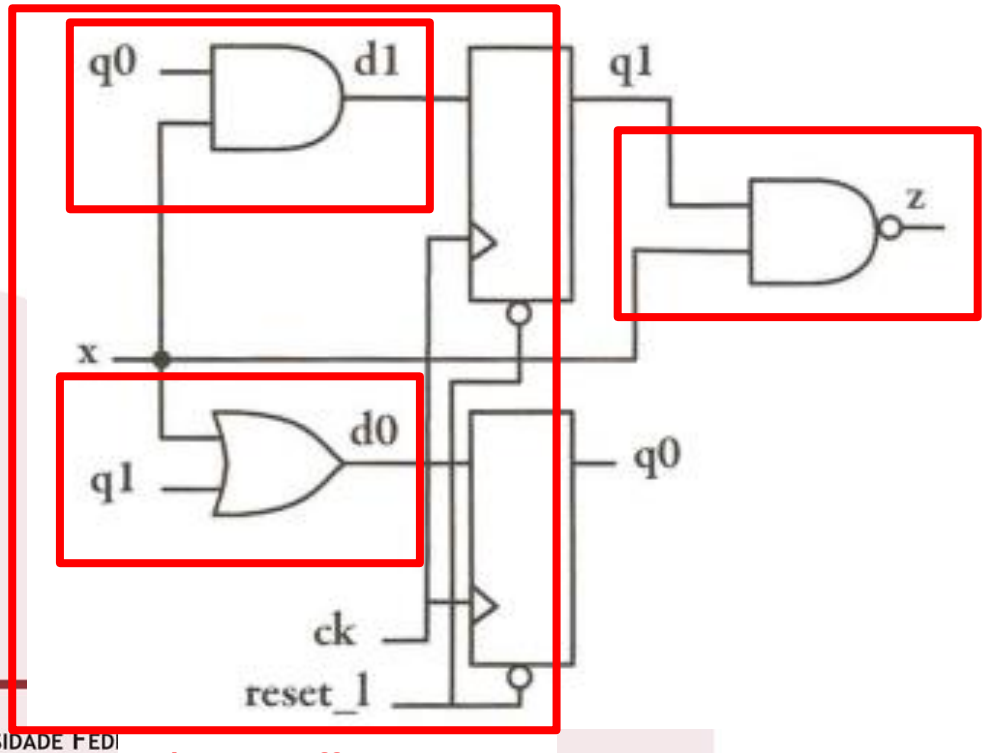


Figure 3.2 — The Karnough Maps



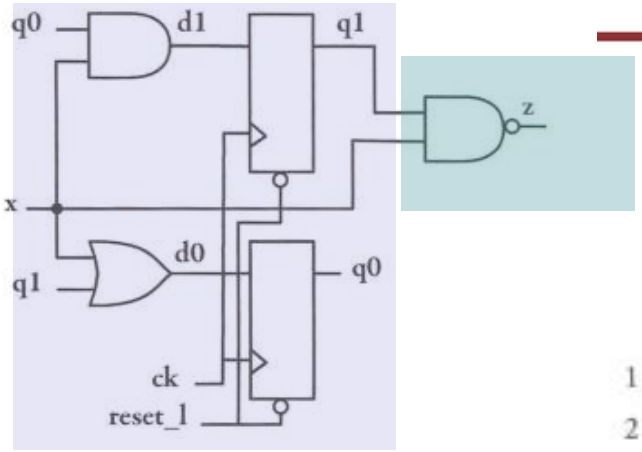


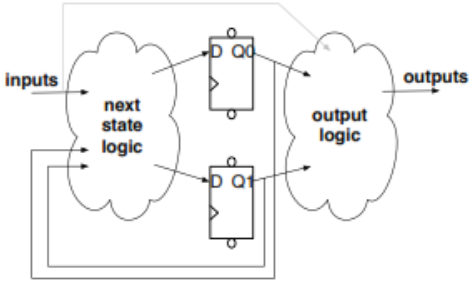


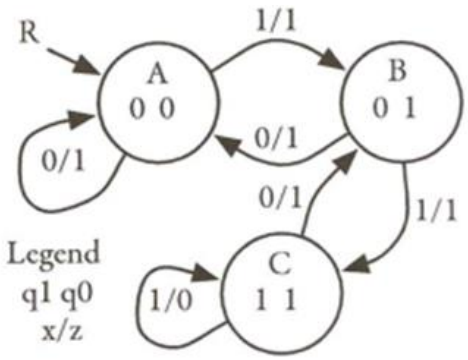
Always_comb ou assign

Always_ff

FSM em SystemVerilog





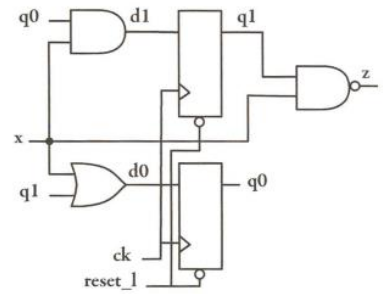


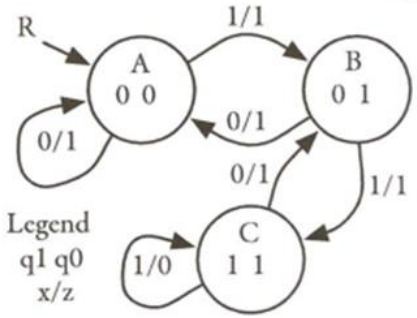
Atribuições concorrentes

```
1   module FSM
2     (input   logic   x, ck, reset_l,
3      output  logic   z);
4     logic   q1, q0;
5
6     always_ff @(posedge ck, negedge reset_l) begin
7       if (~reset_l)
8       {q1, q0} <= 2'b00;
9       else begin
10      q0 <= q1 | x;
11      q1 <= q0 & x;
12      end
13    end
14
15    assign z = ~(x & q1);
16  endmodule: FSM
```

Example 3.7 — A Finite State Machine

FSM em SystemVerilog





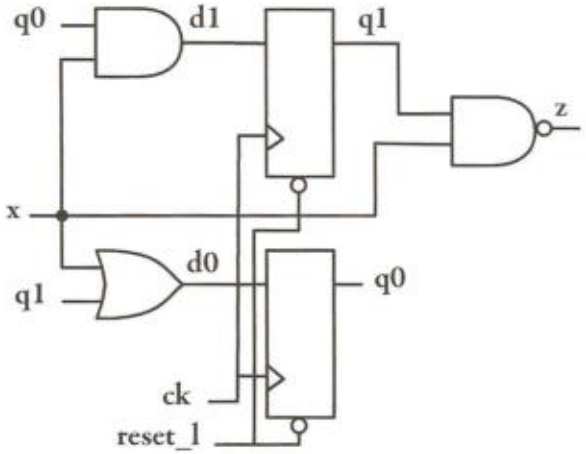

```
1   module FSM_alternate
2     (input logic   x, ck, reset_l,
3      output  logic   z);
4     logic   q1, q0;
5
6     always_ff @(posedge ck, negedge reset_l) l
7       if (~reset_l)
8         {q1, q0} <= 2'b00;
9       else begin
10        q1 <= q0 & x;   // order switched
11        q0 <= q1 | x;
12      end
13    end
14
15    assign z = ~(x & q1);
16  endmodule: FSM_alternate
```

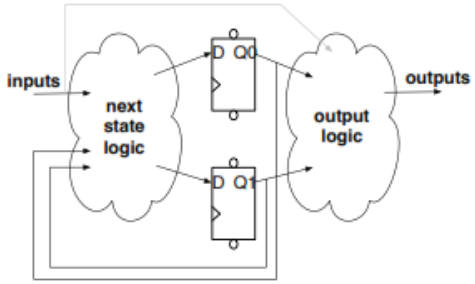
Comportamento similar

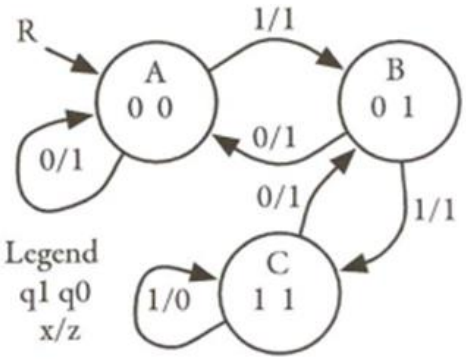
```
1   module FSM
2     (input   logic   x, ck, reset_l,
3      output  logic   z);
4     logic   q1, q0;
5
6     always_ff @(posedge ck, negedge reset_l) begin
7       if (~reset_l)
8         {q1, q0} <= 2'b00;
9       else begin
10        q0 <= q1 | x;
11        q1 <= q0 & x;
12      end
13    end
14
15    assign z = ~(x & q1);
16  endmodule: FSM
```

Example 3.7 — A Finite State Machine

FSM em SystemVerilog







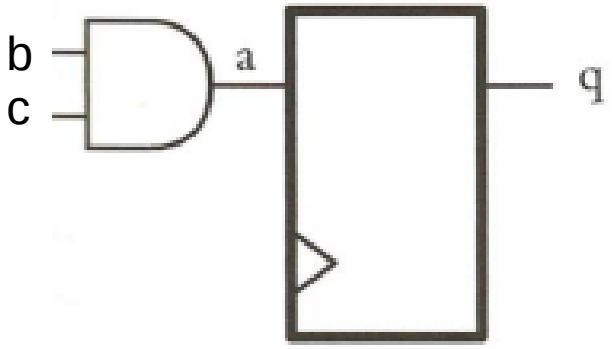

```
1   module FSM_wrong  //Incorrect!
2     (input logic   x, ck, reset_l,
3      output  logic   z);
4     logic   q1, q0;
5
6     always_ff @(posedge ck, negedge reset_l) begin
7       if (~reset_l)
8       {q1, q0} <= 2'b00;
9       else begin
10        q0 = q1 | x;        //no!
11        q1 = q0 & x;        //no!
12      end
13    end
14
15    assign z = ~(x & q1);
16  endmodule: FSM__wrong
```

Atribuições concorrentes

Atribuições não concorrentes

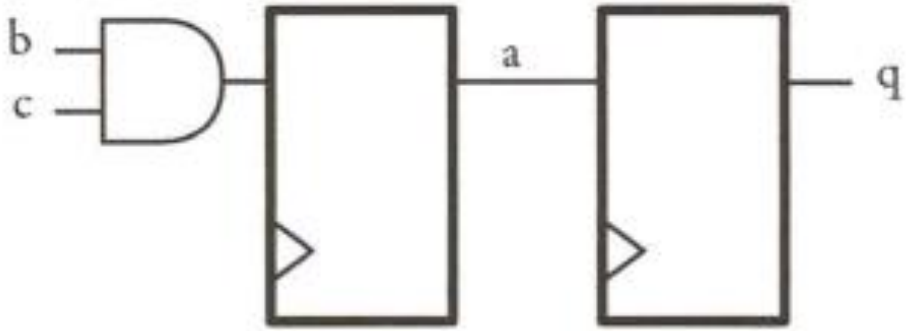
<= vs. =

```
module ff_A
  (output  logic  q,
   input logic  b, c, ck);
  logic  a;

  always_ff (posedge ck) begin
    a = b & c;
    q <= a;
  end
endmodule: ff_A
```



module ff_A

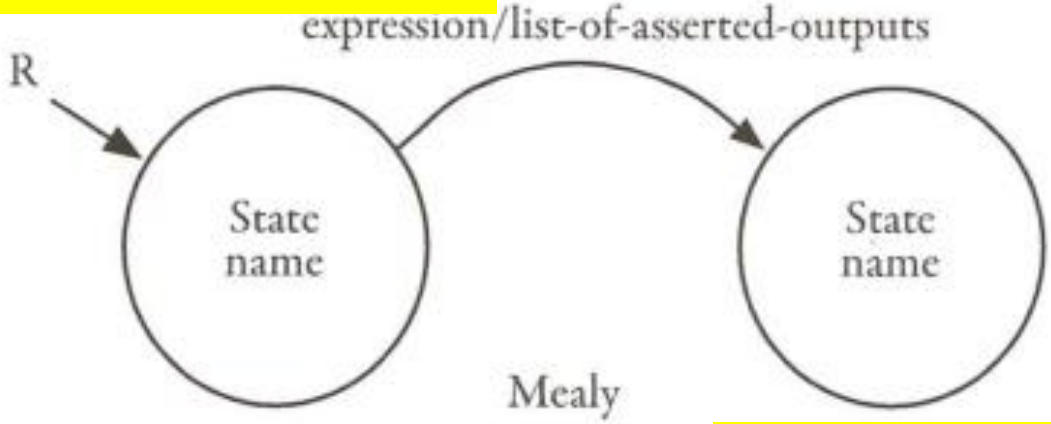
```
module ff_B
  (output  logic  q,
   input   logic  b, c, ck);
  logic  a;

  always_ff (posedge ck) begin
    a <= b & c;
    q <= a;
```



module ff_B

Modelando FSM como Diagramas de Estado

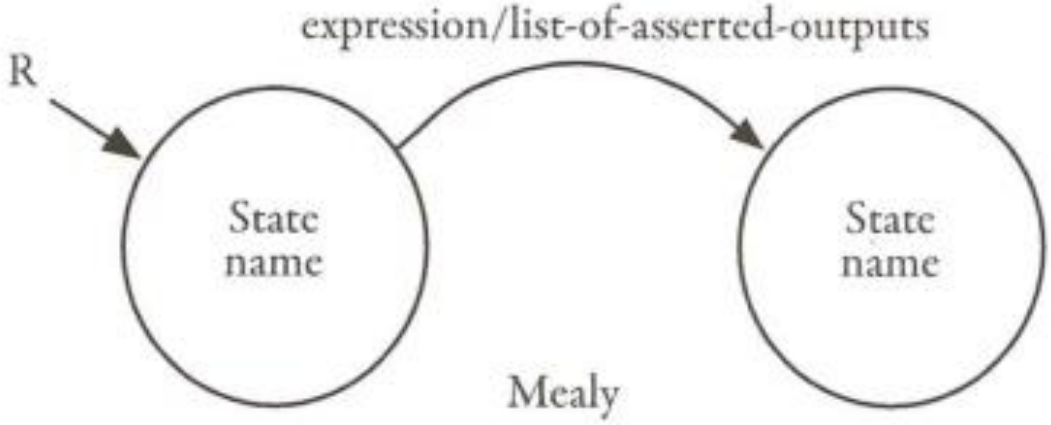
Reset state. One state is always labeled as the reset state using an "R" and an arrow pointing to the reset state.



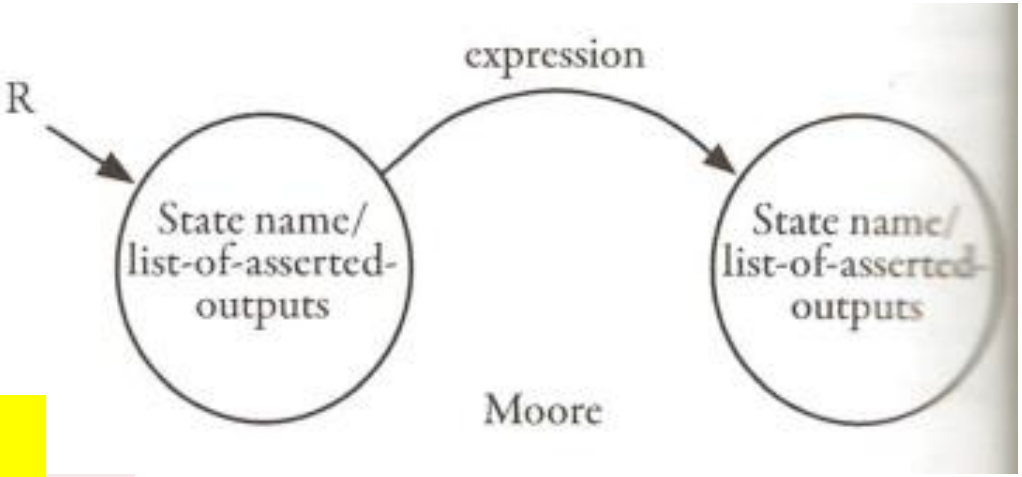
State name. A unique and meaningful name is given to each state. The actual state assignment is normally only shown in the SystemVerilog model.

Transition expression. An Boolean expression is shown by an arc that represents the next state if the expression is TRUE.

Modelando FSM como Diagramas de Estado

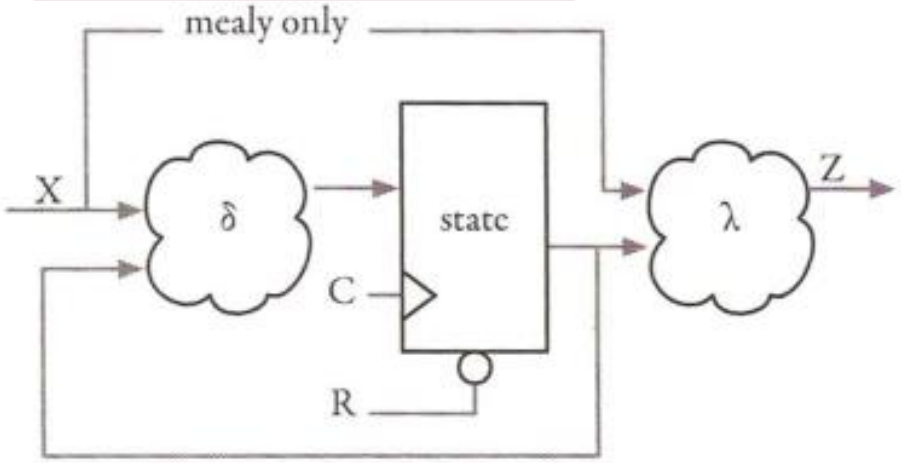


Saídas dependem do estado e da entrada

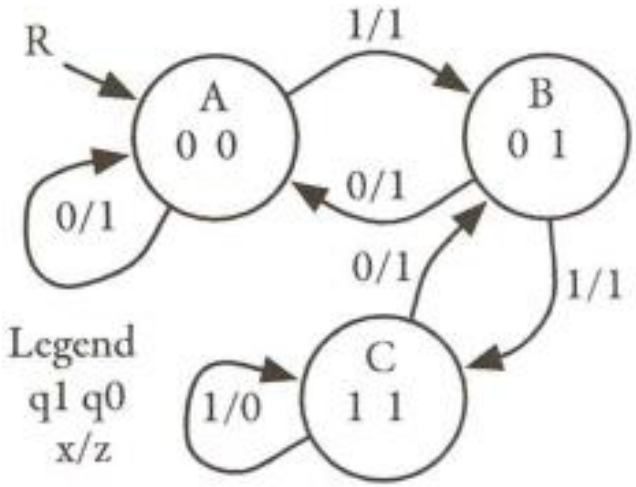


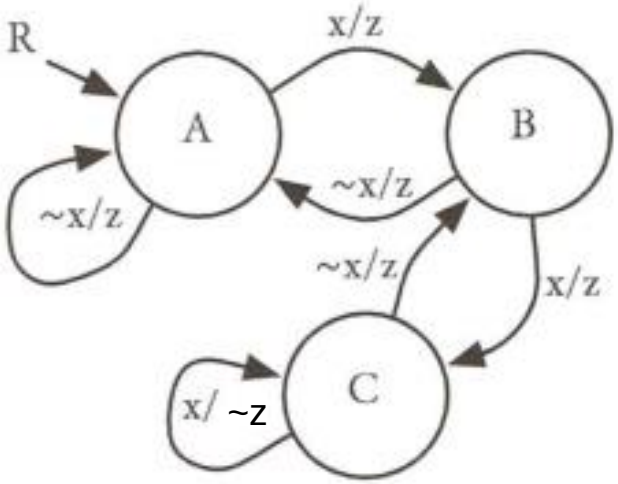
Saídas dependem do estado

Modelando FSM como Diagramas de Estado

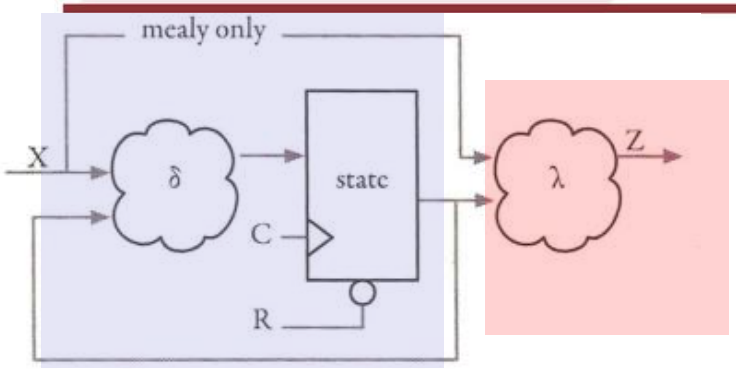


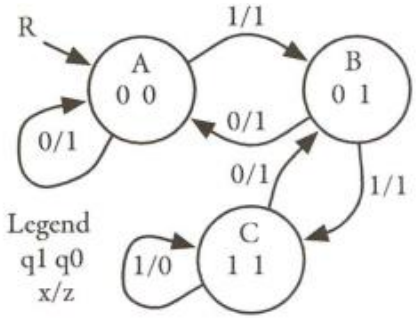
Estados simbólicos e entrada e saída como variáveis

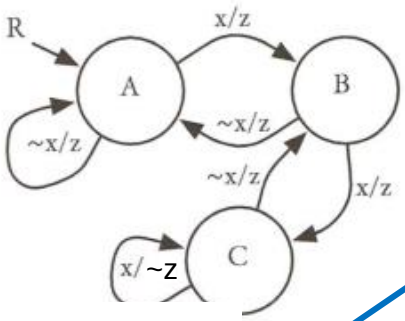




Modelando FSM como Diagramas de Estado





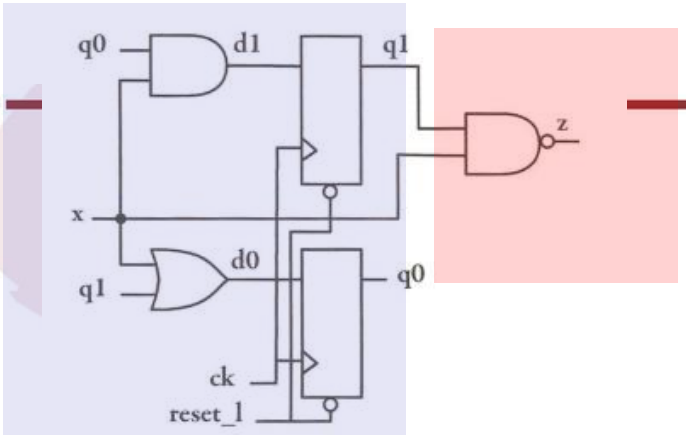


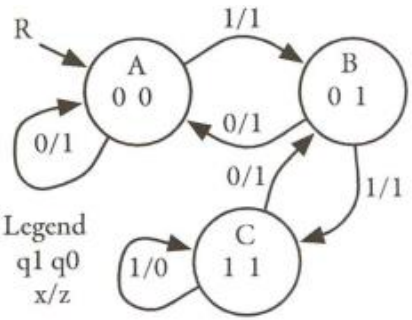
Mudança do estado

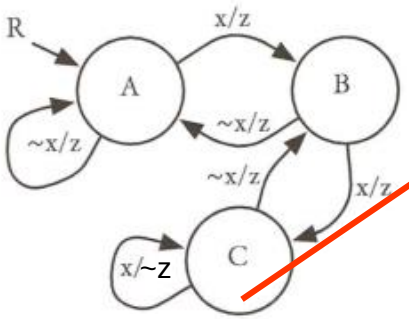
Mudança da saída

```
1   module FSMbehavior
2     (input logic   x, ck, r_l,
3      output  logic   z);
4
5     enum {A, B, C} state; // values differ from
6                           //Figure 3.7
7     always_ff @(posedge ck, negedge r_l) begin
8       if (~r_l)
9         state <= A;
10      else case (state)
11            A: state <= (x) ? B : A;
12            B: state <= (x) ? C : A;
13            C: state <= (x) ? C : B;
14            default: state <= A;
15        endcase
16    end
17    always_comb begin
18      z = 1'b1;
19      if (state == C)   z = ~x;
20    end
21  endmodule: FSMbehavior
```

Modelando FSM como Diagramas de Estado








```
1   module FSMalternate
2     (input   logic   x, ck, r_l,
3      output  logic   z);
4
5     enum {A, B, C} state, nextState;
6     always_ff @(posedge ck, negedge r_l)
7       if (~r_l)  state <= A;
8       else       state <= nextState;
9
10    always_comb
11      case (state)
12        A: nextState = (x) ? B : A;
13        B: nextState = (x) ? C : A;
14        C: nextState = (x) ? C : B;
15        default: nextState = A;
16      endcase
17
18    assign z = (state == C) ? ~x : 1'b1;
19  endmodule: FSMalternate
```

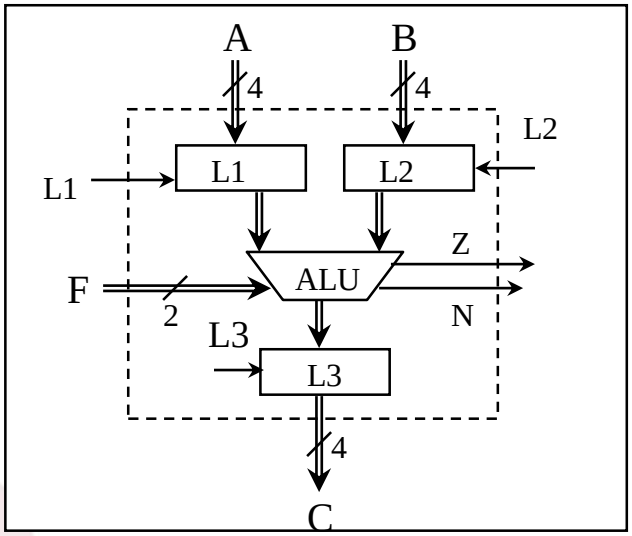
Mudança do estado

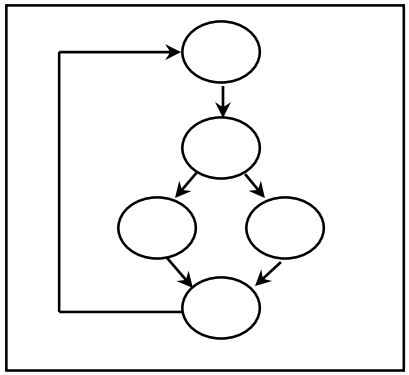
Calculo do próximo estado

Mudança da saída

Projetando um sistema digital

- Estrutura: controle + processamento





Resumo

- Circuitos Combinacionais
  - Always-comb
  - Assign
- Circuitos sequenciais
  - Armazenamento do estado
- Elementos Sequenciais
  - dFF
  - Clock e reset
  - Registrador

Resumo

- Conceitos de Máquinas de Estados
- Modelando FSM usando SystemVerilog
- Atribuições concorrentes
- Modelando Diagramas de Estados em System Verilog
  - FSMs